

Contenido

INTRODUCCIÓN. SQLite	185
Creación y apertura de la BD	185
Acceso a la BD mediante sentencias sql	186
Acceso a la BD mediante código.....	187
Recuperación de la BD.rawQuery	189
Recuperación de la BD con query	190
Creación de una clase BDAdapter.	191
Uso de SimpleCursorAdapter	192
INTRODUCCIÓN. Content Provider	198
Crear un contentprovider	198
Utilizar un contentprovider	205
Utilizar un contentprovider del sistema	207
INTRODUCCIÓN. FileProvider	210
Crear un File provider	210

11.

Acceso a datos. SQLite y ContentProvider

INTRODUCCIÓN. SQLite

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados: la base de datos SQLite y Content Providers.

Vamos a centrarnos en SQLite, aunque no entraremos ni en el diseño de BBDD relacionales ni en el uso avanzado de la BBDD. Para conocer toda la funcionalidad de SQLite se recomienda usar la documentación oficial.

Para el acceso a las bases de datos tendremos tres clases que tenemos que conocer. La clase SQLiteOpenHelper, que encapsula todas las funciones de creación de la base de datos y versiones de la misma. La clase SQLiteDatabase, que incorpora la gestión de tablas y datos. Y por último la clase Cursor, que usaremos para movernos por el recordset que nos devuelve una consulta SELECT.

Creación y apertura de la BD

El método recomendado para la creación de la base de datos es extender la clase SQLiteOpenHelper y sobrescribir los métodos onCreate y onUpgrade. El primer método se utiliza para crear la base de datos por primera vez y el segundo para actualizaciones de la misma. Si la base de datos ya existe y su versión actual coincide con la solicitada, se realizará la conexión a ella.

Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método onUpgrade(). Si la base de datos no existe se llama automáticamente a onCreate().

Para ejecutar la sentencia SQL utilizaremos el método execSQL proporcionado por la API para SQLite de Android.

```
public class BDClientes extends SQLiteOpenHelper {
    String sentencia="create table if not exists clientes (dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);";

    public BDClientes(Context context, String name, SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) { sqLiteDatabase.execSQL(sentencia); }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i2) {
    }
}
```



Una vez creada la clase, instanciaremos un objeto nuevo en la Activity y usaremos uno de los dos siguientes métodos para tener acceso a la gestión de datos: `getWritableDatabase()` para acceso de escritura y `getReadableDatabase()` para acceso de solo lectura. En ambos casos se devolverá un objeto de la clase `SQLiteDatabase` para el acceso a los datos.

```
public class MyActivity extends Activity {
    BDClientes clientes;
    SQLiteDatabase dbClientes;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);
        clientes=new BDClientes(getBaseContext(), "BDCLIENTES", null, 1);
        dbClientes=clientes.getWritableDatabase();
    }
}
```

Como podemos ver en la imagen creamos un objeto del tipo de `BDClientes` con el que vamos a trabajar, llamando al constructor `BDClientes()`, a este método le pasamos el contexto, el nombre de la BBDD, un objeto cursor (con valor null) y la versión de la BD que necesitamos.

En nuestro caso lo que hacemos es ejecutar la creación de las tablas (en este caso solamente una) de la Base de Datos para los clientes. También es necesaria una clase donde esté definida el objeto asociado a la tabla de nuestra BD (el cliente).

Acceso a la BD mediante sentencias sql

El acceso a la BBDD se hace en dos fases, primero se consigue un objeto del tipo `SQLiteDatabase` y posteriormente usamos sus funciones para gestionar los datos. Si hemos abierto correctamente la base de datos entonces podremos empezar con las inserciones, actualizaciones, borrado, etc.

Por ejemplo, vamos a insertar 5 filas en nuestra tabla agenda.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_my);
    clientes=new BDClientes(getBaseContext(), "BDCLIENTES", null, 1);
    dbClientes=clientes.getWritableDatabase();

    if(dbClientes!=null) {
        for(int i=5; i<10; i++) {
            String sentencia="INSERT INTO clientes (dni, nombre, apellidos) VALUES ('"+ i+"', 'nombre'+i+'', 'apellido'+i+'')";
            dbClientes.execSQL(sentencia);
        }

        dbClientes.close();
    }
}
```

Vale, ¿y ahora qué? ¿Dónde está la base de datos que acabamos de crear? ¿Cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente? Vayamos por partes.



En primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

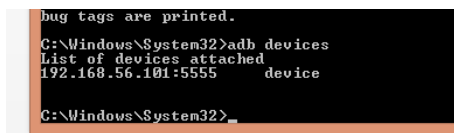
/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos

En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente:

/data/data/ejemplo.basedatos/databases/DBClientes

Para comprobar que los datos se han grabado accedemos de forma remota al emulador a través de su consola de comandos (shell). Para ello, con el emulador abierto, debemos abrir una consola de MS-DOS y utilizar la utilidad adb.exe (Android Debug Bridge) si usamos Genymotion estará situada en C:\Program Files\Genymobile\Genymotion\tools. Si ejecutamos un comando path con esta ruta, podremos acceder al adb desde cualquier posición de directorios (path C:\Program Files\Genymobile\Genymotion\tools).

En primer lugar consultaremos los identificadores de todos los emuladores en ejecución mediante el comando “adb devices”. Esto nos debe devolver una única instancia si sólo tenemos un emulador abierto, que en mi caso particular sería:



```
bug tags are printed.
C:\Windows\System32>adb devices
List of devices attached
192.168.56.101:5555    device
C:\Windows\System32>
```

Tras conocer el identificador de nuestro emulador, vamos a acceder a su shell mediante el comando “adb -s idEmulador shell”. Una vez conectados, ya podemos acceder a nuestra base de datos utilizando el comando sqlite3 pasándole la ruta del fichero, para nuestro ejemplo “sqlite3 /data/data/ejemplo.basedatos/databases/DBClientes”. Si todo ha ido bien, debe aparecernos el prompt de SQLite “sqlite>”, lo que nos indicará que ya podemos escribir las consultas SQL deseadas sobre nuestra base de datos, como por ejemplo un SELECT.

También podemos acudir a la perspectiva DDMS de Android, en la pestaña File Explorer buscar la ruta para comprobar que la BBDD se ha creado.

Con esto ya hemos comprobado que nuestra base de datos se ha creado correctamente, que se han insertado todos los registros de ejemplo y que todo funciona según se espera.

Acceso a la BD mediante código

La API de Android nos ofrece dos métodos para acceder a los datos. El primero de ellos ya lo hemos visto, se trata de ejecutar sentencias SQL a través de execSql. Este método tiene como parámetro una cadena con cualquier instrucción SQL válida.

La otra forma es utilizar los métodos específicos insert(), update() y delete() de la clase SQLiteDatabase. Veamos a continuación cada uno de estos métodos.

insert(): recibe tres parámetros (table, nullColumnHack, values), el primero es el nombre de la tabla, el segundo se utiliza en caso de que necesitemos insertar valores nulos en la tabla



"nullColumnHack" en este caso lo dejaremos pasar ya que no lo vamos a usar y por lo tanto lo ponemos a null y el tercero son los valores del registro a insertar. Los valores a insertar los pasaremos a su vez como elementos de una colección de tipo ContentValues. Estos elementos se almacenan como parejas clave-valor, donde la clave será el nombre de cada campo y el valor será el dato correspondiente a insertar. "

update(): Prácticamente es igual que el anterior método pero con la excepción de que aquí estamos usando el método "update(table, values, whereClause, whereArgs)" para actualizar/modificar registros de nuestra tabla. Este método nos pide el nombre de la tabla "table", los valores a modificar/actualizar "values" (ContentValues), una condición WHERE "whereClause" que nos sirve para indicarle que valor queremos que actualice y como último parámetro "whereArgs" podemos pasarle los valores nuevos a insertar, en este caso no lo vamos a necesitar por lo tanto lo ponemos a null.

delete(): el método "delete(table, whereClause, whereArgs)" nos pide el nombre de la tabla "table", el registro a borrar "whereClause" que tomaremos como referencia su id y como último parámetro "whereArgs" los valores a borrar.

Insertando con ContentValues

```
ContentValues valores=new ContentValues();
valores.put("nombre", "Ana");
valores.put("dni", "11111111");
valores.put("apellidos", "Perez Rico");
dbClientes.insert("clientes",null,valores);
```

```
ContentValues valores=new ContentValues();
valores.put("nombre", "Xavi");
valores.put("dni", "22222111");
valores.put("apellidos", "Perez Rico");
dbClientes.update("clientes",valores,"dni=4",null);
```

Modificando con ContentValues

Borrando con ContentValues

```
String []arg={"1"};
dbClientes.delete("clientes","dni=?",arg);
```

Los valores que hemos dejado anteriormente como null son realmente argumentos que podemos utilizar en la sentencia SQL. Veámoslo con un ejemplo:

```
ContentValues modificar=new ContentValues();
modificar.put("nombre", "Carla");
String[] arg={"6","7"};

dbClientes.update("clientes",modificar,"dni=? OR dni=?",arg);
```

Donde las ? indican los emplazamientos de los argumentos.

- ✚ Crea un ejercicio ejercicioResueltoBD en el que pruebes el código visto anteriormente: Creación de la BD, insertar elementos con SQL y modificar, eliminar e insertar mediante ContentValues.

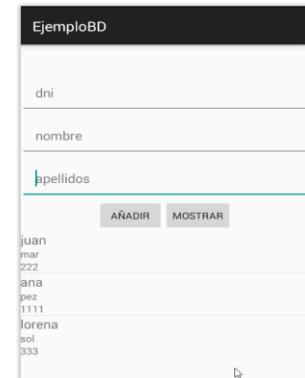


Recuperación de la BD rawQuery

Existen dos formas de recuperar información (SELECT de la base de datos), aunque ambas se apoyan en el concepto de cursor, que es en definitiva el objeto que recoge los resultados de la consulta.

Vamos a **realizar un ejemplo** que nos permita ir insertando registros en la BD y visualizándolos en un listView.

La primera forma de obtener datos es utilizar el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe como parámetro la sentencia SQL, donde se indican los campos a recuperar y los criterios de selección.



```
private boolean seleccionarDatosSelect()
{
    ArrayList<Clientes> listaCliente;
    dbClientes=clientes.getReadableDatabase();
    if(dbClientes!=null) {
        Cursor cursor = dbClientes.rawQuery("select * from clientes order by apellidos", null);
        listaCliente = Clientes.getClientes(cursor);
        dbClientes.close();
        if(listaCliente==null) return false;
        else return true;
    }
    return false;
}
```

Evidentemente, ahora tendremos que recorrer el cursor y visualizar los datos devueltos. Para ello se dispone de los métodos `moveToFirst()`, `moveToNext()`, `moveToLast()`, `moveToPrevious()`, `isFirst()` y `isLast()`.

Existen métodos específicos para la recuperación de datos `getXXX(indice)`, donde XXX indica el tipo de dato (String, Blob, Float,...) y el parámetro índice permite recuperar la columna indicada en el mismo, teniendo en cuenta que comienza en 0.

El método `getClientes` (implementado por nosotros) es el que nos permite leer los datos existentes en la BD y llevarlos al `arrayList`:

```
public static ArrayList<Clientes> getClientes(Cursor cursor)
{
    ArrayList<Clientes> clientes;
    Clientes cliente;
    cursor.moveToFirst();
    if(!cursor.isAfterLast())
    {
        clientes=new ArrayList<Clientes>();
        while(!cursor.isAfterLast())
        {
            cliente=new Clientes(cursor.getString(0),cursor.getString(1),cursor.getString(2));
            clientes.add(cliente);
            cursor.moveToNext();
        }
        return clientes;
    }
    return null;
}
```



Recuperación de la BD con query

La segunda forma de recuperar información de la BD es utilizar el método query(). Recibe como parámetros el nombre de la tabla, un string con los campos a recuperar, un string donde especificar las condiciones del WHERE, otro para los argumentos si los hubiera, otro para el GROUP BY, otro para HAVING y finalmente otro para ORDER BY.

```
private boolean seleccionarDatosCodigo(String[]columnas, String where,String[] valores, String orderBy)
{
    dbClientes=clientes.getReadableDatabase();
    if(dbClientes!=null) {
        Cursor cursor= dbClientes.query("clientes",columnas,where,valores,null,null,orderBy);
        listaCliente = Clientes.getClientes(cursor);
        dbClientes.close();
        if(listaCliente==null) return false;
        else return true;
    }
    return false;
}
```

Donde la llamada al método podría ser la siguiente:

```
seleccionarDatosCodigo(new String[]{"dni", "nombre", "apellidos"}, null, null, "apellidos");
```

- ✚ Crea un ejercicio ejemploBD en el que pruebes el código visto anteriormente: Recuperación de los datos con rawQuery y con query. Para ello crea la aplicación como en la imagen, de forma que con un botón se guarde la información en la BD y con el otro se extraiga y se muestre en una lista o en un recycler. Si vais a utilizar una lista como se muestra en la imagen, deberíeis crear una clase adaptador como la de la siguiente imagen y asociarla a la lista con el setAdapter:

```
lista.setAdapter(new AdaptadorClientes(MyActivity.this,R.layout.list_layout, listaCliente));
```

Clase Adaptador para un ListView:

```
public class AdaptadorClientes extends ArrayAdapter {
    Activity activitycontext;
    ArrayList<Clientes> objects;
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        View vista=convertView;
        if(vista==null)
        {
            LayoutInflater inflater=activitycontext.getLayoutInflater();
            vista=inflater.inflate(R.layout.list_layout, null);
            ((TextView)vista.findViewById(R.id.apellidoList)).setText(objects.get(position).getApellidos());
            ((TextView)vista.findViewById(R.id.nombreList)).setText(objects.get(position).getNombre());
            ((TextView)vista.findViewById(R.id.dniList)).setText(objects.get(position).getDni());
        }
        return vista;
    }
    public AdaptadorClientes(Activity context, int resource, ArrayList objects) {
        super(context, resource, objects);
        activitycontext=context;
        this.objects=objects;
    }
}
```



Creación de una clase BDAdapter.

El modo de programar las aplicaciones, que hemos utilizado en los anteriores ejemplos es poco limpio. Es mucho más conveniente la definición de una clase que contenga toda la información relativa al acceso a la BD y sus métodos. Veamos cómo: (la funcionalidad de la aplicación es la misma)

```
public class BDClientes {
    ClientesOH clientes;
    Activity activity;
    SQLiteDatabase dbClientes;
    ArrayList<Clientes> listaCliente;
    public BDClientes(Activity activity) {
        this.activity=activity;
        clientes=new ClientesOH(activity,"BDCLIENTES",null,1);
    }
    public ArrayList<Clientes> getListaCliente() { return listaCliente; }

    public boolean seleccionarDatosSelect(String sentencia)
    {
        dbClientes=clientes.getReadableDatabase();
        if(dbClientes!=null) {
            Cursor cursor = dbClientes.rawQuery(sentencia, null);
            listaCliente = Clientes.getClientes(cursor);
            dbClientes.close();
            if(listaCliente==null) return false;
            else return true;
        }
        return false; }

    public boolean seleccionarDatosCodigo(String[] columnas, String where,String[] valores, String orderBy)
    {
        dbClientes=clientes.getReadableDatabase();
        if(dbClientes!=null) {
            Cursor cursor= dbClientes.query("clientes",columnas,where,valores,null,null,orderBy);
            listaCliente = Clientes.getClientes(cursor);
            dbClientes.close();
            if(listaCliente==null) return false;
            else return true;
        }
        return false; }
}
```

```
public void insertarDatosCodigo(Clientes cliente)
{
    dbClientes=clientes.getWritableDatabase();
    if(dbClientes!=null) {
        ContentValues valores = new ContentValues();
        valores.put("nombre", cliente.getNombre());
        valores.put("dni", cliente.dni);
        valores.put("apellidos", cliente.apellidos);
        dbClientes.insert("clientes", null, valores);
        dbClientes.close();
    }
}

class ClientesOH extends SQLiteOpenHelper {
    String sentencia = "create table if not exists clientes (dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);";

    public ClientesOH(Context context, String name, SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) { sqLiteDatabase.execSQL(sentencia); }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i2) {

    }
}
```

Como se puede ver en la imagen, la clase BDClientes comprende los métodos necesarios para seleccionar datos o insertarlos, además de la clase interna derivada de SQLiteOpenHelper que nos permite acceder a la BD. Como es natural, podrás crear los métodos necesarios en BDClientes, tanto para eliminar o para hacer cualquiera otra consulta.

Veamos como se crea el objeto en la activity principal y como se hace la llamada a uno de los métodos, en este caso al método seleccionarDatosSelect:




```

BDClientes clientesbd;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_my);

    dniBuscar=(TextView)findViewById(R.id.dni);
    clientesbd=new BDClientes(this);
    clientesbd.seleccionarDatosSelect("select * from clientes where dni>" + dniBuscar.getText().toString() + " order by apellidos");
}

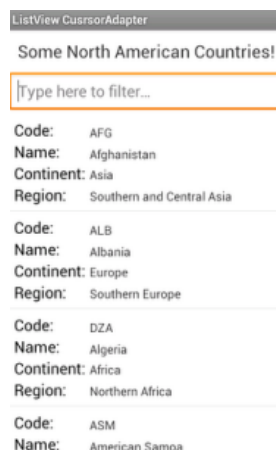
```

Ejercicio Propuesto RecorrerBD.

Uso de SimpleCursorAdapter

Se utiliza para mapear las columnas de un cursor abierto sobre una base de datos, sobre los diferentes elementos visuales contenidos en el control de selección. Es necesario indicar en su definición los siguientes elementos: el contexto, el layout sobre el que se va a definir la vista, el cursor que va a ser origen de los datos, lista de columnas que quiere enlazar, una lista de los campos con los que cada campo es asociado a la vista y un flag (0 por defecto)

Supongamos que en nuestra aplicación tenemos que rellenar un listView con los datos de una consulta que tenemos en un cursor:



Podríamos usar por ejemplo un cursorAdapter para llevar la información directamente del cursor al listView, tal y como se ve a continuación en el siguiente código:

```

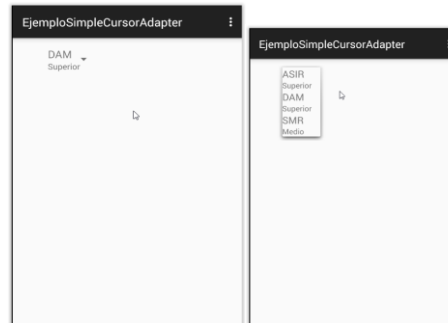
// Columnas contra las que chequea y son las que se quieren enlazar
String[] columns = new String[] { CountriesDbAdapter.KEY_CODE, CountriesDbAdapter.KEY_NAME,
CountriesDbAdapter.KEY_CONTINENT, CountriesDbAdapter.KEY_REGION };
// campos del xml donde asocia
int[] to = new int[] { R.id.code, R.id.name, R.id.continent, R.id.region };
// definición del adaptador
dataAdapter = new SimpleCursorAdapter(this, R.layout.country_info, cursor, columns, to, 0);
ListView listView = (ListView) findViewById(R.id.listView1);
// Assign adapter to ListView
listView.setAdapter(dataAdapter);

```



Con esta definición no es necesario tener definido un arrayList para guardar la solución del cursor.

Vamos a crear un ejemplo para practicar este concepto, será una sencilla actividad con un Spinner que mostrará los módulos con la categoría a la que pertenecen:



Para ello nos creamos la base de datos e insertamos elementos como ya hemos visto hasta el momento:

```
public class MyActivity extends AppCompatActivity {
    SQLiteDatabase sqliteDatabase;
    OHCategoria ohCategoria;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        String[] from = new String[] { "nombre", "cate" };
        int[] to = new int[] { R.id.ciclo, R.id.cate };
        setContentView(R.layout.activity_my);
        ohCategoria=new OHCategoria(this,"BBDcategorias",null,1);
        sqliteDatabase=ohCategoria.getWritableDatabase();
        insertarDatosCodigo();
    }
}
```

```
public void insertarDatosCodigo()
{
    if(sqliteDatabase!=null) {
        ContentValues valores = new ContentValues();
        valores.put("nombre", "ASIR");
        valores.put("cate", "Superior");
        valores.put("idcategoria", 1);
        sqliteDatabase.insert("categoria", null, valores);
        valores.put("nombre", "DAM");
        valores.put("cate", "Superior");
        valores.put("idcategoria", 2);
        sqliteDatabase.insert("categoria", null, valores);
        valores.put("nombre", "SMR");
        valores.put("cate", "Medio");
        valores.put("idcategoria", 3);
        sqliteDatabase.insert("categoria", null, valores);
        sqliteDatabase.close();
    }
}
```



```

class OHCategoria extends SQLiteOpenHelper{

    String cadena="create table if not exists categoria(idcategoria INTEGER PRIMARY KEY NOT NULL, nombre TEXT, cate TEXT);";
    public OHCategoria(Context context, String name, SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) { db.execSQL(cadena); }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    }

}

```

Posteriormente pasamos a crear el cursorAdapter, pero para ello vamos a usar un Spinner para mostrar la información, lo incluimos en el layout principal.

Y por último el SimpleCursorAdapter

```

sqliteDatabase=ohCategoria.getReadableDatabase();
if(sqliteDatabase!=null) {
    Spinner desplegable = (Spinner) this.findViewById(R.id.spinner);
    Cursor cur = sqliteDatabase.rawQuery("select idcategoria as _id, nombre, cate from categoria", null);
    SimpleCursorAdapter mAdapter = new SimpleCursorAdapter(this,R.layout.spinner_layout, cur, from, to, 0x0);
    desplegable.setAdapter(mAdapter);
    sqliteDatabase.close();
}

```

 [Prueba la aplicación del ejemplo anterior](#)

Cursores con RecyclerView

Como era de esperar, no se puede aplicar directamente un adaptador de tipo SimpleCursorAdapter a un tipo RecyclerView. Si queremos aprovechar los beneficios de los cursores en los recyclers, tendremos que fabricárnoslos a medida. Los pasos son los siguientes (veréis que ya os suenan de algo).

Crear una clase Abstracta base que derive de RecyclerView.Adapter y en la que implementaremos los métodos sobrescritos derivados de RecyclerView.Adapter:



```

public abstract class CursorRecyclerViewAdapterAbs extends RecyclerView.Adapter {

    Cursor mCursor;
    public CursorRecyclerViewAdapterAbs(Cursor cursor) { mCursor=cursor;}
    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
        if (mCursor==null) { throw new IllegalStateException("ERROR, cursos vacio");}
        if (!mCursor.moveToPosition(position)) {
            throw new IllegalStateException("ERROR, no se puede encontrar la posicion: " + position);}
        onBindViewHolder(holder, mCursor);}

    public abstract void onBindViewHolder(RecyclerView.ViewHolder holder, Cursor cursor);
    @Override
    public int getItemCount () {
        if (mCursor != null) { return mCursor.getCount();}
        else { return 0; }}
    @Override
    public long getItemId (int position) {
        if(hasStableIds() && mCursor != null){
            if (mCursor.moveToPosition(position)) {
                return mCursor.getLong(mCursor.getColumnIndexOrThrow("_id"));}}
        return RecyclerView.NO_ID;}
}

```

Ahora tendremos que crear nuestra clase RecyclerView derivada de la anterior, en esta manejaremos el cursor extrayendo los elementos y pasándolos al Holder para que los asigne a las vistas correspondientes. Al constructor de esta clase le pasaremos los id de los View donde queremos dejar la información, el cursor, el array con los nombres de los campos y el layout de las líneas del recycles:

```

public class MiRecyclerViewAdapter extends CursorRecyclerViewAdapterAbs {
    private int mLayout;
    private int[] mFrom;
    private int[] mTo;
    public MiRecyclerViewAdapter(int layout, Cursor c, String[] from, int[] to) {
        super(c);
        mLayout = layout;
        mTo = to;
        findColumns(c, from); }
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View v = LayoutInflater.from(parent.getContext())
            .inflate(mLayout, parent, false);
        return new SimpleViewHolder(v,mTo);}
    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, Cursor cursor) {
        ((SimpleViewHolder) holder).bind(0, cursor.getString(mFrom[0]));
        ((SimpleViewHolder) holder).bind(1, cursor.getString(mFrom[1]));
        Bitmap theImage=MyActivity.convertirStringBitmap(cursor.getString(mFrom[2]));
        ((SimpleViewHolder) holder).bind( theImage); }
    private void findColumns(Cursor c, String[] from) {
        if (c != null) {
            int i;
            int count = from.length;
            if (mFrom == null || mFrom.length != count) {
                mFrom = new int[count]; }
            for (i = 0; i < count; i++) {
                mFrom[i] = c.getColumnIndexOrThrow(from[i]); }
            } else { mFrom = null; }
        }
    }
}

```



El Holder será parecido al siguiente:

```
class SimpleViewHolder extends RecyclerView.ViewHolder
{
    TextView[] view=new TextView[2];
    ImageView imagen;


    public SimpleViewHolder (View itemView, int[]to)
    {
        super(itemView);
        view[0]= (TextView) itemView.findViewById(to[0]);
        view[1] = (TextView) itemView.findViewById(to[1]);
        imagen=(ImageView) itemView.findViewById(to[2]);
    }

    public void bind(int pos, String dato)
    {
        view[pos].setText(dato);
    }


    public void bind(Bitmap dato)
    {
        imagen.setImageBitmap(dato);
    }
}
```

Como esta App es un poco diferente a la anterior, se han incluido imágenes para dar más funcionalidad, he creado dos métodos que nos permiten pasar imágenes de Bitmap a String y viceversa para poderlas guardar en la BD a través del cursor. Están localizados en MyActivity y son estáticos para poder acceder a ellos sin necesidad de objeto.


EjemploSimpleCursorAdapter



ASIR
Superior



DAM
Superior



SMR
Medio

```
static public Bitmap convertirStringBitmap(String imagen) {
    byte[] decodedString = Base64.decode(imagen, Base64.DEFAULT);
    return BitmapFactory.decodeByteArray(decodedString, 0, decodedString.length);
}

static public String ConvertirImagenString(Bitmap bitmap) {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    bitmap.compress(Bitmap.CompressFormat.PNG, 90, stream);
    byte[] byte_arr = stream.toByteArray();
    String image_str = Base64.encodeToString(byte_arr, Base64.DEFAULT);
    return image_str;
}
```

InsertarDatos, muy parecida a la anterior pero usando una de las anteriores funciones:



```

public void insertarDatosCodigo() {
    SQLiteDatabase = ohCategoria.getWritableDatabase();
    String img;
    if (sqliteDatabase != null) {
        ContentValues valores = new ContentValues();
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        valores.put("nombre", "ASIR");
        valores.put("cate", "Superior");
        valores.put("idcategoria", 1);
        img = ConvertirImagenString(BitmapFactory.decodeResource(getResources(), R.drawable.ic_adb_black_24dp));
        valores.put("imagen", img);
        SQLiteDatabase.insert("categoria", null, valores);
        valores.put("nombre", "DAM");
        valores.put("cate", "Superior");
        valores.put("idcategoria", 2);
        img = ConvertirImagenString(BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher));
        valores.put("imagen", img);
        SQLiteDatabase.insert("categoria", null, valores);
        valores.put("nombre", "SMR");
        valores.put("cate", "Medio");
        valores.put("idcategoria", 3);
        img = ConvertirImagenString(BitmapFactory.decodeResource(getResources(), R.drawable.ic_local_florist_black_24dp));
        valores.put("imagen", img);
        SQLiteDatabase.insert("categoria", null, valores);
        SQLiteDatabase.close();
    }
}

```

El onCreate de la aplicación principal quedaría así:

```

public class MyActivity extends AppCompatActivity {
    SQLiteDatabase sqLiteDatabase;
    OHCategoria ohCategoria;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        String[] from = new String[] { "nombre", "cate", "imagen" };
        int[] to = new int[] { R.id.ciclo, R.id.cate, R.id.imagen };
        setContentView(R.layout.activity_my);
        ohCategoria = new OHCategoria(this, "BBDCategoria", null, 1);

        insertarDatosCodigo();

        sqLiteDatabase = ohCategoria.getReadableDatabase();
        if (sqLiteDatabase != null) {
            RecyclerView desplegable = (RecyclerView) findViewById(R.id.recycler);
            Cursor cur = sqLiteDatabase.rawQuery("select idcategoria as _id, nombre, cate, imagen from categoria", null);
            MiRecyclerViewAdapter mAdapter = new MiRecyclerViewAdapter(R.layout.recycler_layout, cur, from, to);
            desplegable.setAdapter(mAdapter);
            desplegable.setLayoutManager(new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false));
        }
    }
}

```

- ✚ Prueba la aplicación del ejemplo anterior
- ✚ Adapta la agenda a la BD usando los cursores como los hemos visto en el ejemplo anterior.



INTRODUCCIÓN. Content Provider

Un ContentProvider no es más que el mecanismo proporcionado por la plataforma Android para permitir compartir información entre aplicaciones. Una aplicación que desee que todo o parte de la información que almacena esté disponible de una forma controlada para el resto de aplicaciones del sistema, deberá proporcionar un contentprovider a través del cual se pueda realizar el acceso a dicha información. Este mecanismo es utilizado por muchas de las aplicaciones estándar de un dispositivo Android, como por ejemplo la lista de contactos, la aplicación de SMS, o el calendario/agenda. Esto quiere decir que podríamos acceder a los datos gestionados por estas aplicaciones desde nuestras propias aplicaciones Android haciendo uso de los content providers correspondientes.

Son por tanto dos temas los que debemos tratar en este apartado, por un lado, construir nuevos contentproviders personalizados para nuestras aplicaciones, y por otro utilizar un content provider ya existente para acceder a los datos publicados por otras aplicaciones.

Crear un contentprovider

Para añadir un contentprovider a nuestra aplicación tendremos que:

1. Crear una nueva clase que extienda a la clase android **ContentProvider**.
2. Declarar el nuevo *contentprovider* en nuestro fichero AndroidManifest.xml

Por supuesto nuestra aplicación tendrá que contar previamente con algún método de almacenamiento interno para la información que queremos compartir. Lo más común será disponer de una base de datos SQLite, por lo que será esto lo que utilizaré para todos los ejemplos de este apartado, pero internamente podríamos tener los datos almacenados de cualquier otra forma, por ejemplo en ficheros de texto, ficheros XML, etc. El contentprovider será el mecanismo que nos permita publicar esos datos a terceros de una forma homogénea y a través de una interfaz estandarizada.

Un primer detalle a tener en cuenta es que los registros de datos proporcionados por un contentprovider deben contar siempre con un campo llamado `_ID` que los identifique de forma unívoca del resto de registros. Como ejemplo, los registros devueltos por un contentprovider de clientes podría tener este aspecto:

<code>_ID</code>	Ciente	Telefono	Email
3	Antonio	900123456	email1@correo.com
7	Jose	900123123	email2@correo.com
9	Luis	900123987	email3@correo.com

Sabiendo esto, es interesante que nuestros datos también cuenten internamente con este campo `_ID` (no tiene por qué llamarse igual) de forma que nos sea más sencillo después generar los resultados del contentprovider.



Aplicación Ejemplo

Vamos a construir una aplicación de ejemplo con una base de datos SQLite que almacene los datos de una serie de clientes con una estructura similar a la tabla anterior

1. La aplicación principal de ejemplo no mostrará, en principio, nada en pantalla ni hará nada con la información, ya que no nos va a interesar el tratamiento directo de los datos por parte de la aplicación principal, sino su utilización a través del content provider. Crearemos nuestra aplicación con una base de datos SQLite, para ello usaremos la técnica que ya conocemos:

```
public class ClientesSqliteHelper extends SQLiteOpenHelper {
    String cadena="CREATE TABLE Clientes " +
        "( _id INTEGER PRIMARY KEY AUTOINCREMENT, " +
        " nombre TEXT, " +
        " telefono TEXT, " +
        " email TEXT );";

    public ClientesSqliteHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String nombre, telefono, email;
        db.execSQL(cadena);
        //Insertamos 15 clientes de ejemplo
        for(int i=1; i<=15; i++)
        {
            nombre = "Cliente" + i; telefono = "900-123-00" + i; email = "email" + i + "@email.com";
            db.execSQL("INSERT INTO Clientes (nombre, telefono, email) " + "VALUES ('" + nombre + "', '" + telefono + "', '" + email + "');");
        }
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS Clientes");
        db.execSQL(cadena);
    }
}
```

Para simplificar el ejemplo, el método **onUpgrade()** se limita a eliminar la tabla actual y crear una nueva con la nueva estructura.

2. construir el nuevo content provider que permitirá compartir los datos de nuestra aplicación con otras. El acceso a un content provider se realiza siempre mediante una URI, similar a:

"content://tema11.xusa.proyectos.ejemplocontentprovider/clientes"



Las direcciones URI de los content providers están formadas por 3 partes:

- i. el prefijo "**content://**" que indica que dicho recurso deberá ser tratado por un content provider.
- ii. el identificador en sí del content provider, también llamado *authority*. Dado que este dato debe ser único es una buena práctica utilizar como authority el nombre del paquete + el nombre del content provider".
- iii. Por último, se indica la entidad concreta a la que queremos acceder dentro de los datos que proporciona el content provider. En nuestro caso será la tabla de "clientes" ya que es la única existente, pero dado que un content provider puede contener los datos de varias entidades distintas en este último tramo de la URI habrá que especificarlo. Indicar por último que en una URI se puede hacer referencia directamente a un registro concreto de la entidad seleccionada. Esto se haría indicando al final de la URI el ID de dicho registro. Por ejemplo la uri "**content://tema11.xusa.proyectos.ejemplocontentprovider/clientes/23**" haría referencia directa al cliente con `_ID = 23`.

El siguiente paso será extender la clase ContentProvider. Si echamos un vistazo a los métodos abstractos que tendremos que implementar veremos que tenemos los siguientes:

- `onCreate()` → inicializa todos los recursos necesarios para el funcionamiento del nuevo content provider.
- `query()`, `insert()`, `update()`, `delete()` → métodos que permiten acceder a los datos consulta, inserción, modificación y eliminación, respectivamente.
- `getType()` → permite conocer el tipo de datos devuelto por el content provider.

Además de implementar estos métodos, también definiremos una serie de constantes dentro de nuestra nueva clase provider, que ayudarán posteriormente a su utilización. Veamos este paso a paso. Vamos a crear una nueva clase **CientesProvider** que extienda de **ContentProvider**.

Lo primero que vamos a definir es la URI con la que se accederá a nuestro content provider y encapsularla en un objeto de tipo Uri.



```
String uri = "content://temall.xusa.proyectos.ejemplocontentprovider/clientes";
Uri CONTENT_URI = Uri.parse(uri);
```

A continuación vamos a definir las constantes para los datos proporcionados por nuestro content provider. Las columnas predefinidas que deben tener todos los content providers, por ejemplo la columna `_ID` están definidas en la clase **BaseColumns**, por lo que definiremos una clase, interna a content provider, tomando como base esta clase para añadir las columnas. *Sería buena práctica crear, en nuestro content provider, una librería con la clase que deriva de **BaseColumn** y que tiene los nombres de las columnas. Esto serviría para que al incluir la librería en la aplicación de acceso a content provider, podríamos tener acceso a estos nombres sin posibilidad de error.*

```
//Clase interna para declarar las constantes de columna
public class Clientes implements BaseColumns
{
    public static final String COL_NOMBRE = "nombre", COL_TELEFONO = "telefono", COL_EMAIL = "email";
    static String uri = "content://temall.xusa.proyectos.ejemplocontentprovider/clientes";
    public static Uri CONTENT_URI = Uri.parse(uri);
}
```

La primera tarea que nuestro content provider deberá hacer cuando se acceda a él será interpretar la URI utilizada. Para facilitar esta tarea Android proporciona una clase llamada **UriMatcher**, capaz de interpretar determinados patrones en una URI. Para conseguir esto definiremos un objeto **UriMatcher** y dos nuevas constantes que representen los dos tipos de URI que hemos indicado:

```
public class ContentProviderClientes extends ContentProvider {
    String uri = "content://temall.xusa.proyectos.ejemplocontentprovider/clientes";
    Uri CONTENT_URI = Uri.parse(uri);
    final int CLIENTES = 1, CLIENTES_ID = 2;
    String TABLA_CLIENTES = "clientes";
    private ClientesSqliteHelper bDSH;
    UriMatcher uriMatcher;

    @Override
    public boolean onCreate() {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("temall.xusa.proyectos.ejemplocontentprovider", "clientes", CLIENTES);
        uriMatcher.addURI("temall.xusa.proyectos.ejemplocontentprovider", "clientes/#", CLIENTES_ID);
        bDSH= new ClientesSqliteHelper(getContext(), "bdClientes", null, 1);

        return false;
    }
}
```

Método **query()** → Este método recibe como parámetros: una **URI**, una lista de **nombres de columna**, un **criterio de selección**, una **lista de valores para las variables** utilizadas en el criterio



anterior, y un **criterio de ordenación**. Todos estos datos son análogos a los de la consulta de datos en SQLite para Android.

Para distinguir entre los dos tipos de URI posibles utilizaremos como ya hemos indicado el objeto **uriMatcher**, utilizando su método **match()**. Si el tipo devuelto es **CLIENTES_ID**, es decir, que se trata de un acceso a un cliente concreto, sustituiremos el criterio de selección por uno que acceda a la tabla de clientes sólo por el ID indicado en la URI. Para obtener este ID utilizaremos el método **getLastPathSegment()** del objeto uri, que extrae el último elemento de la URI, en este caso el ID del cliente.

Hecho esto, ya tan sólo queda realizar la consulta a la base de datos mediante el método **query()** de **SQLiteDatabase**.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
    //Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if(uriMatcher.match(uri) == CLIENTES_ID) where = "_id=" + uri.getLastPathSegment();
    SQLiteDatabase db = bDSH.getReadableDatabase(); //getWritableDatabase();
    Cursor c = db.query(TABLA_CLIENTES, projection, where, selectionArgs, null, null, sortOrder);
    return c;
}
```

Como vemos, los resultados se devuelven en forma de objeto **Cursor**, una vez más exactamente igual a como lo hace el método **query()** de **SQLiteDatabase**.

Por su parte, los métodos **update()** y **delete()** → son completamente análogos a éste, con la única diferencia de que devuelven el número de registros afectados en vez de un cursor a los resultados. Vemos directamente el código:



```

@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
    int cont;
    String where = selection;
    if(uriMatcher.match(uri) == CLIENTES_ID) where = "_id=" + uri.getLastPathSegment();
    SQLiteDatabase db = bDSH.getWritableDatabase();
    cont = db.update(TABLA_CLIENTES, values, where, selectionArgs);

    return cont;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    int cont;
    String where = selection;

    if(uriMatcher.match(uri) == CLIENTES_ID) where = "_id=" + uri.getLastPathSegment();
    SQLiteDatabase db = bDSH.getWritableDatabase();
    cont = db.delete(TABLA_CLIENTES, where, selectionArgs);

    return cont;
}

```

El método **insert()** → sí es algo diferente, aunque igual de sencillo. La diferencia en este caso radica en que debe devolver la URI que hace referencia al nuevo registro insertado. Para ello, obtendremos el nuevo ID del elemento insertado como resultado del método **insert()** de **SQLiteDatabase**, y posteriormente construiremos la nueva URI mediante el método auxiliar **ContentUri.withAppendedId()** que recibe como parámetro la URI de nuestro content provider y el ID del nuevo elemento.

```

@Override
public Uri insert(Uri uri, ContentValues values) {
    long regId = 1;
    SQLiteDatabase db = bDSH.getWritableDatabase();
    regId = db.insert(TABLA_CLIENTES, null, values);
    Uri newUri = ContentUri.withAppendedId(CONTENT_URI, regId);

    return newUri;
}

```

Por último, el método **getType()** → Este método se utiliza para identificar el tipo de datos que devuelve el content provider. Este tipo de datos se expresará como un *MIME Type*, al igual que hacen los navegadores web para determinar el tipo de datos que están recibiendo tras una petición a un servidor. Identificar el tipo de datos que devuelve un content provider ayudará por ejemplo a Android a determinar qué aplicaciones son capaces de procesar dichos datos.



Existen dos tipos MIME distintos para cada entidad del content provider, uno de ellos destinado a cuando se devuelve una lista de registros como resultado, y otro para cuando se devuelve un registro único concreto. De esta forma, seguiremos los siguientes patrones para definir uno y otro tipo de datos:

- "vnd.android.cursor.item/vnd.xxxxxx" → Registro único
- "vnd.android.cursor.dir/vnd.xxxxxx" → Lista de registros

En mi caso de ejemplo, he definido los siguientes tipos:

- "vnd.android.cursor.item/vnd.xusa.proyectos.ejemplocontentprovider"
- "vnd.android.cursor.dir/vnd.xusa.proyectos.ejemplocontentprovider "

Con esto en cuenta, la implementación del método **getType()** quedaría como sigue:

```
@Override
public String getType(Uri uri) {
    int match = uriMatcher.match(uri);

    switch (match) {
        case CLIENTES:
            return "vnd.android.cursor.dir/vnd.xusa.proyectos.ejemplocontentprovider";
        case CLIENTES_ID:
            return "vnd.android.cursor.item/vnd.xusa.proyectos.ejemplocontentprovider";
        default:
            throw new IllegalArgumentException("Unsupported URI: " + uri);
    }
}
```

Con esto ya hemos completado la implementación del nuevo contentprovider. Pero aún nos queda un paso más, como indicamos al principio del apartado. Debemos declarar el content provider en nuestro fichero **AndroidManifest.xml** de forma que una vez instalada la aplicación en el dispositivo Android conozca la existencia de dicho recurso.

```
<provider android:name=".ContentProviderClientes"
    android:authorities="temall.xusa.proyectos.ejemplocontentprovider"/>

</application>
```

En el siguiente apartado veremos cómo utilizar este nuevo contentprovider para acceder a los datos de nuestra aplicación de ejemplo, y también veremos cómo podemos utilizar alguno de los content provider predefinidos por Android para consultar datos del sistema.

🔧 Crea una App que permita compartir una tabla Clientes como la vista en el anterior código, sigue los pasos del ejemplo para crear el Content Provider.



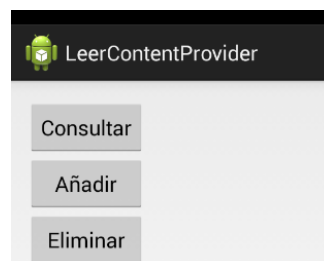
Utilizar un contentprovider

Primero vamos a aprender a hacer uso de nuestro contentprovider para acceder a los datos de clientes, desde otra aplicación. Además, veremos cómo también podemos acceder a datos del propio sistema Android (logs de llamadas, lista de contactos, agenda telefónica, bandeja de entrada de sms, etc) utilizando este mismo mecanismo.

Utilizar un contentprovider ya existente es muy sencillo, sobre todo comparado con el laborioso proceso de construcción de uno nuevo. Para comenzar, debemos obtener una referencia a un **Content Resolver**, objeto a través del que realizaremos todas las acciones necesarias sobre el contentprovider. Esto es tan fácil como utilizar el método **getContentResolver()** desde nuestra actividad para obtener la referencia indicada. Una vez obtenida la referencia al content resolver, podremos utilizar sus métodos **query()**, **update()**, **insert()** y **delete()** para realizar las acciones equivalentes sobre el contentprovider.

Seguimos con la aplicación Ejemplo

Por ver varios ejemplos de la utilización de estos métodos crearemos una nueva aplicación de ejemplo, LeerContentProvider, con tres botones en la pantalla principal, uno para hacer una consulta de todos los clientes, otro para insertar registros nuevos, y el último para eliminar todos los registros nuevos insertados con el segundo botón.



1. Consulta de clientes. El procedimiento será prácticamente igual al de acceso a bases de datos SQLite). Comenzaremos por definir un array con los nombres de las columnas de la tabla que queremos recuperar en los resultados de la consulta, que en nuestro caso serán el ID, el nombre, el teléfono y el email. Si hemos seguido la recomendación que se dice en punto anterior, de crear la librería con la clase derivada de BaseColumn. Ahora deberemos incluir la librería, para de esta forma tener acceso a estos nombres sin posibilidad de error.

Consulta de todos los registros de la tabla:



```
private Cursor consultar() {
    String[] projection = new String[]{Clientes._ID, Clientes.COL_NOMBRE, Clientes.COL_TELEFONO, Clientes.COL_EMAIL};
    cr = getContentResolver();
    Cursor cur = cr.query(Clientes.CONTENT_URI, projection, //Columnas a devolver
        null, //Condición de la query
        null, //Argumentos variables de la query
        null); //Orden de los resultados

    return cur;
}
```

Consulta de los registros de la tabla que cumplan la condición de selection:

```
private Cursor consultar() {
    String[] projection = new String[]{Clientes._ID, Clientes.COL_NOMBRE, Clientes.COL_TELEFONO, Clientes.COL_EMAIL};
    cr = getContentResolver();
    String selection = "(" + Clientes.COL_NOMBRE + " = ?) AND ("
        + Clientes.COL_TELEFONO + " = ?)";
    String[] selectionArgs = new String[] {"Cliente6", "900-123-006"};
    //Hacemos la consulta
    Cursor cur = cr.query(Clientes.CONTENT_URI, projection, //Columnas a devolver
        selection, //Condición de la query
        selectionArgs, //Argumentos variables de la query
        null); //Orden de los resultados
    return cur;
}
```

Recorrer el cursor para meter los datos en un editText:

```
void recorrerCursor(Cursor cur)
{
    cur.moveToFirst();
    texto.setText("");
    do
    {
        texto.setText( texto.getText().toString() + cur.getString(cur.getColumnIndex(Clientes.COL_NOMBRE)) + " " +
            cur.getString(cur.getColumnIndex(Clientes.COL_TELEFONO)) + "\n");
        cur.moveToNext();
    }while(!cur.isLast());
    texto.setText( texto.getText().toString() + cur.getString(cur.getColumnIndex(Clientes.COL_NOMBRE)) + " " +
        cur.getString(cur.getColumnIndex(Clientes.COL_TELEFONO)) + "\n");
}
```

- Para insertar nuevos registros, el trabajo será también exactamente igual al que se hace al tratar directamente con bases de datos SQLite. Rellenaremos en primer lugar un objeto ContentValues con los datos del nuevo cliente y posteriormente utilizamos el método insert() pasándole la URI del contentprovider en primer lugar, y los datos del nuevo registro como segundo parámetro.

```
private void insertar()
{
    cr = getContentResolver();
    ContentValues x=new ContentValues();
    x.put(Clientes.COL_NOMBRE, "ANA ANITA");
    x.put(Clientes.COL_TELEFONO, "4444444444");
    x.put(Clientes.COL_EMAIL, "adfaff@adfadfa");
    cr.insert(Clientes.CONTENT_URI, x);
}
```



3. Por último, y más sencillo todavía, la eliminación de registros la haremos directamente utilizando el método `delete()` del content resolver, indicando como segundo parámetro el criterio de localización de los registros que queremos eliminar, que en este caso serán los que hayamos insertado nuevos con el segundo botón de ejemplo (aquellos con nombre = 'ANA ANITA').

```
private void eliminar()
{
    cr = getContentResolver();
    cr.delete(Clientes.CONTENT_URI, Clientes.COL_NOMBRE + " = 'ANA ANITA'", null);
}
```

✚ Crea la aplicación `LeerContentProvider` con los tres botones que te permitan realizar la consulta, la inserción y la eliminación del código anterior.

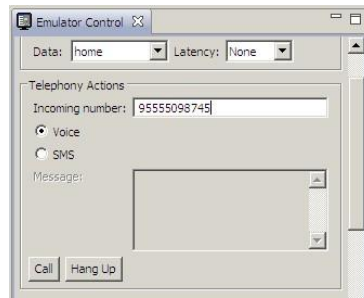
Utilizar un contentprovider del sistema

Después de haber visto el ejemplo anterior es igual de sencillo acceder a los proveedores de contenido de la propia plataforma Android. En la documentación oficial del paquete `android.provider` podemos consultar los datos que tenemos disponibles a través de este mecanismo, entre ellos encontramos por ejemplo: el historial de llamadas, la agenda de contactos y teléfonos, las bibliotecas multimedia (audio y video), o el historial y la lista de favoritos del navegador.

Por ver un ejemplo de acceso a este tipo de datos, vamos a realizar una consulta al historial de llamadas del dispositivo, para lo que accederemos al contentprovider **`android.provider.CallLog`**.

En primer lugar vamos a registrar varias llamadas en el emulador de Android, de forma que los resultados de la consulta al historial de llamadas contengan algunos registros. Haremos por ejemplo varias llamadas salientes desde el emulador y simularemos varias llamadas entrantes desde el DDMS. Las primeras son sencillas, simplemente ve al emulador, accede al teléfono, marca y descuelga igual que lo harías en un dispositivo físico. Y para emular llamadas entrantes podremos hacerlo accediendo a la vista del DDMS. En esta vista, si accedemos a la sección "*Emulator Control*" veremos un apartado llamado "*Telephony Actions*". Desde éste, podemos introducir un número de teléfono origen cualquiera y pulsar el botón "*Call*" para conseguir que nuestro emulador reciba una llamada entrante. Sin aceptar la llamada en el emulador pulsaremos "*Hang Up*" para terminar la llamada simulando así una llamada perdida.





Hecho esto, procedemos a realizar la consulta al historial de llamadas utilizando el content provider indicado, y para ello crearemos una nueva aplicación a la que le añadiremos un botón Mostrar Llamadas.

Consultando la documentación del content provider veremos que podemos extraer diferentes datos relacionados con la lista de llamadas. Nosotros nos quedaremos sólo con dos significativos, el número origen o destino de la llamada, y el tipo de llamada (entrante, saliente, perdida). Los nombres de estas columnas se almacenan en las constantes **Calls.NUMBER** y **Calls.TYPE** respectivamente.

Decidido esto, actuaremos igual que antes. Definiremos el array con las columnas que queremos recuperar, obtendremos la referencia al content resolver y ejecutaremos la consulta llamando al método **query()**. Por último, recorreremos el cursor obtenido y procesamos los resultados. Igual que antes, lo único que haremos será escribir los resultados al cuadro de texto situado bajo los botones. Veamos el código:

```
String[] projection = new String[] { CallLog.Calls.TYPE, CallLog.Calls.NUMBER };
Uri llamadasUri = CallLog.Calls.CONTENT_URI;
ContentResolver cr = getContentResolver();
int tipo, colTipo, colTelefono;
String tipoLlamada, telefono;

Cursor cur = cr.query(llamadasUri, projection, null, null, null);
if (cur.moveToFirst())
{
    tipoLlamada = "";
    colTipo = cur.getColumnIndex(CallLog.Calls.TYPE);
    colTelefono = cur.getColumnIndex(CallLog.Calls.NUMBER);
    texto.setText("");
    do{
        tipo = cur.getInt(colTipo);
        telefono = cur.getString(colTelefono);
        if(tipo == CallLog.Calls.INCOMING_TYPE) tipoLlamada = "ENTRADA";
        else if(tipo == CallLog.Calls.OUTGOING_TYPE) tipoLlamada = "SALIDA";
        else if(tipo == CallLog.Calls.MISSED_TYPE) tipoLlamada = "PERDIDA";
        texto.append(tipoLlamada + " - " + telefono + "\n");
    } while (cur.moveToNext());
}
```

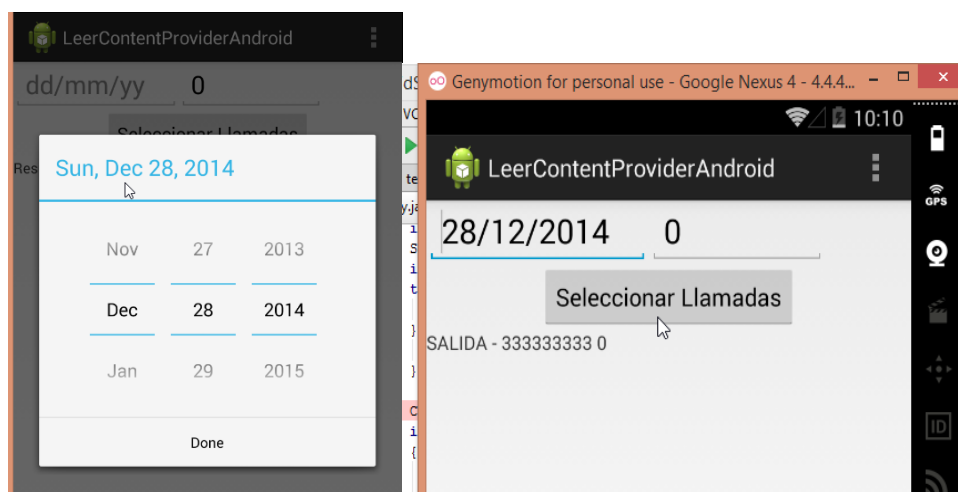
Lo único fuera de lo normal que hacemos en el código anterior es la decodificación del valor del tipo de llamada recuperado, que la hacemos comparando el resultado con las constantes **Calls.INCOMING_TYPE** (entrante), **Calls.OUTGOING_TYPE** (saliente), **Calls.MISSED_TYPE** (perdida) proporcionadas por la propia clase provider.



Un último detalle importante. Para que nuestra aplicación pueda acceder al historial de llamadas del dispositivo tendremos que incluir en el fichero AndroidManifest.xml los permisos READ_CONTACTS y READ_CALL_LOG utilizando la cláusula <uses-permission> correspondiente.

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.READ_CALL_LOG"/>
```

✚ Modifica el anterior ejemplo de manera que aparezca un par de editText, el primero con la fecha y el segundo con la duración de las llamadas, con esta información se hará una selección de los registros que se mostrarán (los de esa fecha y con igual o mayor duración). Si pulsamos el editText de la fecha de forma larga nos mostrará un dialogo con el calendario, que nos permitirá seleccionar la fecha, otra opción será escribir la fecha correctamente en el EditText, en el caso en que la fecha sea incorrecta se mostrarán los registros con la fecha actual y en editText se mostrará también esa fecha.



✚ En este ejercicio usaremos el ejemplo de CrearContentProvider (de principio de tema) y también consultaremos el ContentProvider de Contactos del sistema de nuestro dispositivo, tendremos que leer por un lado la información de los contactos de nuestro dispositivo (nombre, telefono y email) y pasarlo a la base de datos Clientes, usando el ContentProvider que hemos construido en el ejemplo, para probar su buen funcionamiento podemos usar la aplicación de ejemplo LeerContentProvider, que nos mostrará todos los elementos insertados en la base de datos. Podríamos usar un ListView para mostrar los resultados

Tendremos que tener en cuenta que para leer los contactos del sistema, deberemos acceder a la clase ContactsContract.Contacts, pero los teléfonos y email son gestionados por las siguientes y respectivas clases ContactsContract.CommonDataKinds.Phone y ContactsContract.CommonDataKinds.Email.



INTRODUCCIÓN. FileProvider

De igual forma que podemos permitir que otras aplicaciones compartan datos de la BD de nuestra app a través de un contentprovider, también podemos compartir, solamente, algún fichero que creamos necesarios, para ello podemos usar el mecanismo de FileProvider.

Crear un File provider

Preparar la actividad que comparte archivos

Para ofrecer un archivo de forma segura desde nuestra aplicación a otra, necesitamos usar la clase **FileProvider** derivada de **Contentprovider**.

La clase FileProvider es parte de la Librería de Soporte v4. Para definir un FileProvider para nuestra aplicación necesitaremos modificar nuestro manifiesto añadiendo una entrada <provider> dentro de <application>, que especificará algunos elementos:

- la autoridad a usar en la generación de las URIs de contenido.
- nombre de un archivo XML que especifique los directorios que queremos que comparta nuestra aplicación.
- Otros atributos que especifiquen permisos.

Ejemplo:

```
<provider
  android:name="android.support.v4.content.FileProvider"
  android:authorities="com.mydomain.fileprovider"
  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data
    android:name="android.support.FILE_PROVIDER_PATHS"
    android:resource="@xml/file_paths" />
</provider>
```

En este ejemplo, el atributo **android:authorities** especifica la autoridad de URL que queremos usar para las URIs, para este elemento seguiremos la misma forma de construcción que la usada en la autoridad del ContentProvider. En el ejemplo, la autoridad es com.mydomain.fileprovider. Para nuestra propia aplicación, especificaremos una autoridad que consiste en el valor de **android:package** de la aplicación con la string "fileprovider" añadido al mismo. Para aprender más acerca de estos valores véase [provider](#).

El elemento hijo <meta-data> del <provider> apunta a un archivo XML que especifica los directorios que queremos compartir. El atributo android:resource es la ruta y nombre del archivo, sin la extensión .xml. Para crear correctamente este archivo deberemos crear el



archivo **filepaths.xml** en el subdirectorio **res/xml/** de nuestro proyecto. El siguiente ejemplo muestra el contenido de **res/xml/filepaths.xml**.

```
<paths xmlns:android="http://schemas.android.com/apk/res/android">
  <files-path name="my_images" path="images/" />
  <files-path name="my_docs" path="docs/" />
</paths>
```

En este ejemplo, la etiqueta **<files-path>** comparte directorios dentro del directorio **files/** del **almacenamiento interno** de nuestra aplicación, podremos añadir tantas etiquetas **<files-path>** como directorios necesitemos compartir. En el ejemplo el atributo **path** comparte el subdirectorio **images/** y el subdirectorio **doc/** de **files/**. El atributo **name** le dice a **FileProvider** que agregue el segmento de ruta **myimages** a las URIs de contenido para los archivos en el subdirectorio **images/** de **files**.

Además del elemento **<files-path>**, podemos usar el elemento **<external-path>** para compartir directorios en un almacenamiento externo, y el elemento **<cache-path>** para compartir directorios en nuestro directorio de caché interno. Para aprender más, véase la documentación para [FileProvider](#).

Nota: El archivo XML es la única manera de especificar los directorios que queremos compartir, no podemos agregar un directorio programáticamente.

Cuando nuestra aplicación genera una URI de contenido para un archivo, ésta contiene la autoridad especificada en el elemento **<provider>** **"com.mydomain.fileprovider"**, la ruta **my_images/**, y el nombre del archivo. Por ejemplo, en caso de solicitar un archivo **default_image.jpg** para el ejemplo anterior, su uri de contenido sería:

content://com. mydomain.fileprovider/my_images/default_image.jpg

Crear la activity que prepara el intent

Cuando una aplicación decide compartir un archivo deberá tener una actividad que prepare un intent específico, que será el que se utilice para mandar la Uri de contenido al resto de aplicaciones que quieran su archivo.

```
File imagePath = new File(this.getFilesDir(), "default_image.jpg");

Intent mResultIntent = new Intent("com.mydomain.ACTION_RETURN_FILE");
setResult(Activity.RESULT_CANCELED, null);
Uri fileUri = FileProvider.getUriForFile(this, "com.mydomain.fileprovider", imagePath);
if (fileUri != null) {
    mResultIntent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    mResultIntent.setDataAndType(fileUri, getContentResolver().getType(fileUri));
    this.setResult(Activity.RESULT_OK, mResultIntent);
} else {
    mResultIntent.setDataAndType(null, "");
    this.setResult(RESULT_CANCELED, mResultIntent);
}
```



Definiremos una subclase de Activity que obtendrá un objeto File para el nombre del archivo seleccionado y lo pasaremos como argumento a `getUriForFile()`, junto con la autoridad que hayamos especificado en el elemento `<provider>` para el FileProvider. Recordemos que solo podemos generar URIs de contenido para los archivos que residen en un directorio que hayamos especificado en el archivo meta-data que contiene el elemento `<paths>`.

Ahora que tenemos una URI, necesitamos permitir a la aplicación cliente acceder al archivo. Para permitirle el acceso, le concederemos permisos, agregando la URI de contenido a un Intent, y luego estableceremos marcas de permiso en el Intent. Los permisos que concedamos son temporales y expiran automáticamente cuando se elimina la pila de tareas de la aplicación receptora. Para compartir el archivo con la aplicación que lo solicita, pasaremos el Intent a `setResult()`. Cuando la Activity que hemos definido se haya eliminado, el sistema enviará el Intent que contiene la URI de contenido a la aplicación cliente. Por lo tanto, **debemos cerrar la activity definida una vez haya realizado los pasos anteriores.**

Como es normal al crear una Activity nueva deberemos especificarla en el Manifest, pero para que sea arrancable desde otra aplicación se deberán añadir los siguientes elementos: un filtro de intent que coincida con la acción `ACTION_PICK` y las categorías `CATEGORY_DEFAULT` y `CATEGORY_OPENABLE`. Añadiremos también filtros de tipos MIME para los archivos que nuestra aplicación entregue a otras aplicaciones. El siguiente ejemplo muestra cómo especificar la nueva Activity y el filtro de intent:

```
...
<activity
    android:name=".FileSelectActivity"
    android:label="@string/File_Selector" >
    <intent-filter>
        <action
            android:name="android.intent.action.PICK"/>
        <category
            android:name="android.intent.category.DEFAULT"/>
        <category
            android:name="android.intent.category.OPENABLE"/>
        <data android:mimeType="text/plain"/>
        <data android:mimeType="image/*"/>
    </intent-filter>
</activity>
```

Actividad que solicita el Archivo

Para solicitar un archivo a la aplicación servidora, la aplicación cliente llama a **`startActivityForResult`** con un Intent que contiene la acción, como **`ACTION_PICK`**, y el tipo MIME que la aplicación cliente puede manejar.

La aplicación servidora envía de vuelta la URI de contenido del archivo a la aplicación cliente en un Intent. Este Intent se pasa a la aplicación cliente al sobrescribir `onActivityResult()`. Una vez que la aplicación cliente tiene la URI de contenido del archivo, puede acceder al archivo, obteniendo su `FileDescriptor`.

El siguiente ejemplo muestra cómo la aplicación cliente maneja el Intent enviado desde la aplicación servidora, y cómo la aplicación cliente obtiene el `FileDescriptor` usando la URI de contenido:



```

Intent mRequestFileIntent = new Intent(Intent.ACTION_PICK);
mRequestFileIntent.setType("text/plain");
startActivityForResult(mRequestFileIntent, 0);
.....
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode != RESULT_OK) return;
    else {
        Uri returnUrl = data.getData();
        try {
            parcelFileDescriptor = getContentResolver().openFileDescriptor(returnUrl, "r");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            Log.e("MainActivity", "File not found.");
            return;
        }
        FileDescriptor fd = parcelFileDescriptor.getFileDescriptor();
    }
}

```

El método **openFileDescriptor()** devuelve un **ParcelFileDescriptor** para el archivo. Desde este objeto, la aplicación cliente obtiene un objeto **FileDescriptor**, el cual puede usar ahora para leer el archivo.

Este proceso es muy seguro, pues la URI de contenido es la única parte de datos que la aplicación cliente recibe. Puesto que esta URI no contiene una ruta de directorio, la aplicación cliente no puede descubrir ni abrir ningún otro archivo de la aplicación servidora. Solo la aplicación cliente accede al archivo, y lo hace solamente con los permisos concedidos por la aplicación servidora. Los permisos son temporales, por lo que, una vez que la pila de tareas de la aplicación cliente se elimina, ya no se podrá acceder al archivo desde otra aplicación.

🔧 Usando el ejercicio LeerEscribirXML del tema anterior al respecto. Vas a crear una actividad nueva que se ejecutará cuando otra aplicación la solicite y compartirá el archivo `datos.xml`. La aplicación solicitante deberá leer el archivo recibido y mostrarlo por pantalla.

